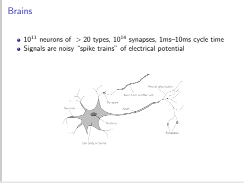
AI #12

└Outline

Outline

- Brains
- Neural networks
- Perceptrons
- Multilayer perceptrons
- Applications of neural networks

Today we will be looking at neural networks. This is the basis of the famed deep learning. We first deal with the brain – which motivated this subject – and the neurons in it. Then we shift our focus of study to the corresponding perceptron. We will see that each perceptron has its limitations, but if we connect enough together, we can achieve almost everything. This is the essence of the theory behind deep learning. Finally, we examine a specific application and its effectiveness.



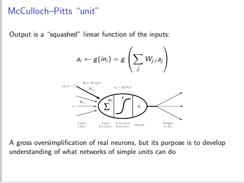
Brains

- 10^{11} neurons of > 20 types, 10^{14} synapses, 1ms–10ms cycle time
- Signals are noisy “spike trains” of electrical potential

The brain is made up of neurons. A neuron is a brain cell that collects, processes, and propagates electrical signals. Signals are transmitted through synapses, this is the connection of neurons.

AI #12

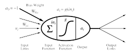
└McCulloch–Pitts “unit”



McCulloch–Pitts “unit”

Output is a “squashed” linear function of the inputs:

$$a_i \leftarrow g(in_i) = g\left(\sum_j W_{j,i} a_j\right)$$

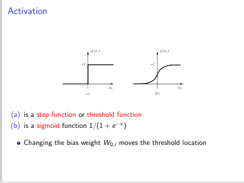


A gross oversimplification of real neurons, but its purpose is to develop understanding of what networks of simple units can do

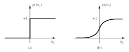
In the 1940s, a simple mathematical model of the neuron was completed, which can be seen here. Since then, more accurate models have been developed for brain studies, but this simple model can be used well at solving artificial intelligence tasks.

The neural network can be treated as a directed graph. The vertices here are named as “unit”s and correspond to a neuron each. Weighted edges indicate connections between neurons. The weighted sum of the activations (a_j) arriving at the vertex is taken, and then the activation function is applied to it, and this gives the output of the unit.

An addition – according to which there is an imaginary input of $a_0 = -1$ and a corresponding weight – proved to be effective.



Activation

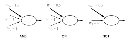


(a) is a step function or threshold function
(b) is a sigmoid function $1/(1+e^{-x})$

- Changing the bias weight $W_{0,i}$ moves the threshold location

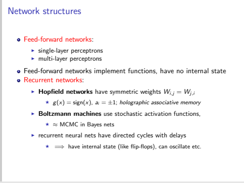
What do we expect from our unit, the perceptron? If we get "good" results, the output should be 1 (or close to it); and accordingly if we get a "bad" input, the output should be 0 (or a nearby number). This can be achieved with a sign function or with a step function, but we will see later that the differentiability of the function is useful for us. The linear activation function is not recommended, because the combination of these kinds of functions will also be a linear function, which we cannot use to approach all functions. Therefore, in addition to the sigmoid function shown in the figure, tanh is the most common activation function today.

Implementing logical functions



- McCulloch and Pitts: every Boolean function can be implemented

In the 1940s it was proven that the basic logic functions can implemented with these units, so by properly connecting them any logic circuit could be implemented! All we need to do is move the activation function left and right, which we can do with W_0. Notice that the negation is solved using $W_1 = -1$!

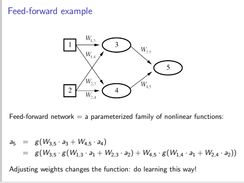


Network structures

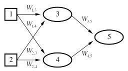
- Feed-forward networks:
  - single-layer perceptrons
  - multi-layer perceptrons
- Feed-forward networks implement functions, have no internal state
- Recurrent networks:
  - **Hopfield networks** have symmetric weights $W_{i,j} = W_{j,i}$
    - $g(x) = \text{sign}(x)$, $a_i = \pm 1$; *holographic associative memory*
  - **Boltzmann machines** use stochastic activation functions,
    - $\approx$ MCMC in Bayes nets
  - recurrent neural nets have directed cycles with delays
    - $\implies$ have internal state (like flip-flops), can oscillate etc.

How these perceptrons be connected? One option is to only move forward, the edges pointing from left to right (feed-forward network). Then we could substitute each value a_i into a function of the additional perceptrons and finally get a piece of a very complicated function. That is, the output is a function of the current input, i.e. the network has no state/memory.

In case of a recurrent grid, it is possible that the output of the perceptron is one of its own inputs. With this, short-term memory can be realized, but our model becomes significantly more complicated.



Feed-forward example



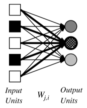
Feed-forward network = a parameterized family of nonlinear functions:

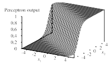
$$\begin{aligned} a_5 &= g(W_{3,5} \cdot a_3 + W_{4,5} \cdot a_4) \\ &= g(W_{3,5} \cdot g(W_{1,3} \cdot a_1 + W_{2,3} \cdot a_2) + W_{4,5} \cdot g(W_{1,4} \cdot a_1 + W_{2,4} \cdot a_2)) \end{aligned}$$

Adjusting weights changes the function: do learning this way!

Consider a very simple graph. The output of node 5 is obtained from the formula. Substituting the occurrences of a_3 and a_4 by their functions, we get a formula in which only input 1 and input 2 are included, so in reality the output will be a function of these. On the other hand, if we change the weights that were treated as constants in the function until now, we get other functions, so the weights are the parameters of the resulting function. This change is the basis of learning. The question is: can we change the weights in such a way that we get the function we need?

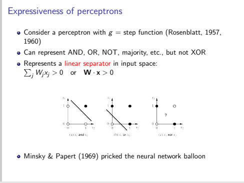
Single-layer perceptrons





- Output units all operate separately—no shared weights
- Adjusting weights moves the location, orientation, and steepness of cliff

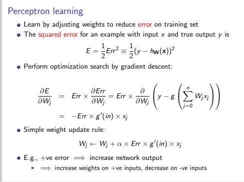
The perceptrons of feed-forward networks are usually organized into layers, each layer receiving input signals only from the previous layer, i.e. the perceptrons of a given layer are independent of each other. A special version of this network is where there is only one layer. Then each perceptron is independent of all the others. By changing the weights and parameters, we can specify which input to take into account, so we can modify the function shown on the right: shift, rotate, stretch (within certain limits).


Expressiveness of perceptrons

- Consider a perceptron with g = step function (Rosenblatt, 1957, 1960)
- Can represent AND, OR, NOT, majority, etc., but not XOR
- Represents a linear separator in input space: $\sum_j W_j x_j > 0$ or $\mathbf{W} \cdot \mathbf{x} > 0$
- Minsky & Papert (1969) pricked the neural network balloon

Let g be the step function. We have already seen that the AND, OR, NOT function can be expressed with a single perceptron. Similarly, it can be seen that this holds for the function majority. However, the XOR function cannot be expressed. Why not? The activation function fires when its argument is greater than 0. Its argument is a weighted sum that can be rewritten into a vector product. It turns out that the perceptron essentially acts as a linear separator, i.e. it fires on one side of the hyperplane, and not on the other side. In the case of AND, OR and NOT, the cases of true and false can be easily separated by a hyperplane. In the case of XOR, however, such a plane does not exist.

Nevertheless, perceptrons should not be considered useless. The majority function can be easily expressed with a perceptron, and even teaching this function to a precepton can be done very quickly. A decision diagram for the same task would be very large, and even its teaching would not yield encouraging results.



Perceptron learning

- Learn by adjusting weights to reduce error on training set
- The squared error for an example with input x and true output y is

$$E = \frac{1}{2}Err^2 \equiv \frac{1}{2}(y - h_{\mathbf{W}}(\mathbf{x}))^2$$

- Perform optimization search by gradient descent:

$$\begin{aligned}\frac{\partial E}{\partial W_j} &= Err \times \frac{\partial Err}{\partial W_j} = Err \times \frac{\partial}{\partial W_j}\left(y - g\left(\sum_{j=0}^{n} W_j x_j\right)\right) \\ &= -Err \times g'(in) \times x_j\end{aligned}$$

- Simple weight update rule:

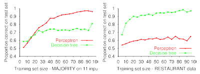
$$W_j \leftarrow W_j + \alpha \times Err \times g'(in) \times x_j$$

- E.g., $+$ve error $\implies$ increase network output
  - $\implies$ increase weights on $+$ve inputs, decrease on -ve inputs

“Teaching” here means trying to minimize the error measured on the teaching set by choosing the appropriate parameters (weights). So consider the n-dimensional space given by the parameters and find the point where the value of the error function is minimal. To do this, calculate the sum of squares of the difference between the results obtained and those expected. Derivatives according to parameters help to move in the best direction. Each weight should be updated using the formula shown here.

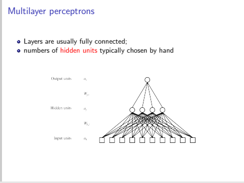
Perceptron learning contd.

- Perceptron learning rule converges to a consistent function
- *for any linearly separable data set*



- Perceptron learns majority function easily, DTL is hopeless
- DTL learns restaurant function easily, perceptron cannot represent it

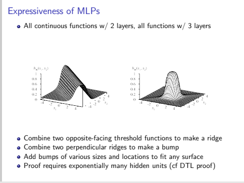
The two examples shown here illustrate how a decision diagram and a perceptron perform. The perception learns the majority function very quickly, while the decision diagram is not able to learn it. In the case of the restaurant problem mentioned earlier, the perceptron can't learn it because this task is not linearly separable.



Multilayer perceptrons

- Layers are usually fully connected;
- numbers of hidden units typically chosen by hand

In case of multilayer neural networks, each pair of perceptrons of two adjacent levels is connected. The number of perceptrons of intermediate (hidden) levels is usually set manually, there is no universal recipe for all problems.

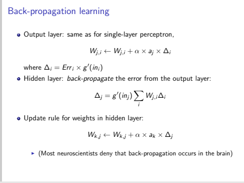


Expressiveness of MLPs

- All continuous functions w/ 2 layers, all functions w/ 3 layers
- Combine two opposite-facing threshold functions to make a ridge
- Combine two perpendicular ridges to make a bump
- Add bumps of various sizes and locations to fit any surface
- Proof requires exponentially many hidden units (cf DTL proof)

Any continuous function can be described by a two-layer network, and any function by a three-layer network. Be careful, because this mathematical result does not mention the number of perceptrons in the hidden layer, which could be infinite!

As you can see, a ridge can be formed with two functions, and a bump can be formed from two ridges. And with a sufficient number and appropriate size of humps, any function can be approximated.



Back-propagation learning

- Output layer: same as for single-layer perceptron,

$$W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times \Delta_i$$

where $\Delta_i = Err_i \times g'(in_i)$

- Hidden layer: *back-propagate* the error from the output layer:

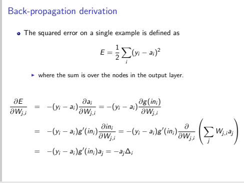
$$\Delta_j = g'(in_j) \sum_i W_{j,i} \Delta_i$$

- Update rule for weights in hidden layer:

$$W_{k,j} \leftarrow W_{k,j} + \alpha \times a_k \times \Delta_j$$

  - (Most neuroscientists deny that back-propagation occurs in the brain)

If we work with a multi-layer neural network, the last layer is taught the same way as for perceptrons. We introduce the concept of modified error (Δ_i). In case of hidden layers, we have no information about what inputs the elements of the teaching set will have and what the output will be, so we can only estimate these quantities from the output layer. However, the formula will be similar to the previous one.



Back-propagation derivation

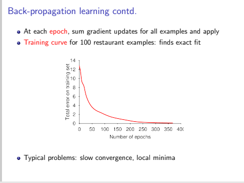
- The squared error on a single example is defined as

$$E = \frac{1}{2}\sum_i (y_i - a_i)^2$$

  - where the sum is over the nodes in the output layer.

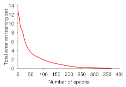
$$\begin{aligned}
\frac{\partial E}{\partial W_{j,i}} &= -(y_i - a_i)\frac{\partial a_i}{\partial W_{j,i}} = -(y_i - a_i)\frac{\partial g(in_i)}{\partial W_{j,i}} \\
&= -(y_i - a_i)g'(in_i)\frac{\partial in_i}{\partial W_{j,i}} = -(y_i - a_i)g'(in_i)\frac{\partial}{\partial W_{j,i}}\left(\sum_j W_{j,i}a_j\right) \\
&= -(y_i - a_i)g'(in_i)a_j = -a_j\Delta_i
\end{aligned}$$

If the output layer contains more than one perceptron, then the errors for each perceptron are added together to give the total error. With the help of the derivatives belonging to the given weight we can distribute this error amongst the perceptrons of the previous layer (back-propagation) and we can update the weights of the edges. In the previous layer, by processing these errors, we can go a step back the same way, and so on.



Back-propagation learning contd.

- At each epoch, sum gradient updates for all examples and apply
- Training curve for 100 restaurant examples: finds exact fit



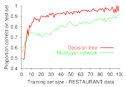
- Typical problems: slow convergence, local minima

Consider a graph where the intermediate hidden layer contains 4 perceptrons. It can be seen from the figure that the network has found the perfect solution with a training set containing 100 examples.

Nevertheless, the method is not omnipotent, convergence is sometimes very slow and may even get stuck at local extremes.

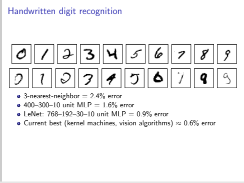
Back-propagation learning contd.

- Learning curve for MLP with 4 hidden units:



- MLPs are quite good for complex pattern recognition tasks,
  - but resulting hypotheses cannot be understood easily

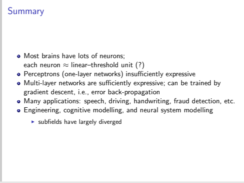
Here you can see the effectiveness of teaching the neural network and the decision diagram in case of the restaurant example. Neural networks are usually used for complex pattern recognition.



Handwritten digit recognition

- 3-nearest-neighbor = 2.4% error
- 400–300–10 unit MLP = 1.6% error
- LeNet: 768–192–30–10 unit MLP = 0.9% error
- Current best (kernel machines, vision algorithms) ≈ 0.6% error

An example of pattern recognition is the recognition of handwritten digits. A public dataset with thousands of digits is freely available. The top row contains digits that are easy to recognize, while the bottom row contains problematic digits. Several methods can be used to solve this problem. The kNN method can be used as a classification problem, where $k = 3$. A three-layer neural network – its parameters (number of perceptrons per level) is given in its name – gave a better result. A four-layer neural network achieved even better results. We can also see in comparison what was the best result 15 years ago with a more complicated network.



Summary

- Most brains have lots of neurons;
  each neuron $\approx$ linear–threshold unit (?)
- Perceptrons (one-layer networks) insufficiently expressive
- Multi-layer networks are sufficiently expressive; can be trained by gradient descent, i.e., error back-propagation
- Many applications: speech, driving, handwriting, fraud detection, etc.
- Engineering, cognitive modelling, and neural system modelling
  - subfields have largely diverged

Our brains contain a lot of neurons, which we now treat as a linear separator. A perceptron (the mathematical model of the neuron) is not expressive enough on its own, but if we connect a lot of these, it becomes very capable for a range of purposes. We can think of this neural network as a function that can be fine-tuned by choosing its parameters (weights). The common teaching method is back-propagation. The neural network is used in many places and is becoming more prevalent. Although the teaching is very resource intensive, the use of the finished neural network is very simple, it is even actively used by some of the apps found on our mobile phones.